Towards Good C++ Interval Libraries: Tricks AND Traits

Frédéric Goualard

Accepted for publication at: *4th Asian Symposium on Computer Mathematics (ASCM'2000), Thailand*

TOWARDS GOOD C++ INTERVAL LIBRARIES: TRICKS AND TRAITS

FRÉDÉRIC GOUALARD
Institut de Recherche en Informatique de Nantes
2, rue de la Houssinière
F-44322 NANTES CEDEX 3
E-mail: Frederic.Goualard@irin.univ-nantes.fr

Despite its usefulness for overcoming floating-point arithmetic defects, manipulating imprecise data, and performing non-linear global optimization, interval arithmetic is not yet a first-class type in any computer language. Some attempts are underway for Fortran and Java, not for C++. However, some add-on C++ interval libraries already exist though they do not respect the C++ standard library spirit since they are not generic (one cannot choose the type for the interval bounds). The design and implementation of JAIL, a C++ templated library based on the "traits" C++ programming technique, are presented in this paper; parameterization is shown to enhance the reliability, the versatily, and the portability of the library across multiple platforms. The impact on efficiency of parameterization through templated classes and of some implementation tricks is also investigated.

1 Introduction

Following *Moore's Law*, computers performances nearly double every 18 months. But, due to the recourse to floating-point arithmetic for numerical programs, it only means they compute plain wrong results faster and faster. In the sixties, another Moore [1] conceived of *interval arithmetic* as a means to compute with floats while controlling the accuracy of the results. Roughly speaking, interval arithmetic consists in replacing any real constant by an interval containing it, using extensions of real operations to intervals, and by preserving the inclusion of any intermediate result through outward rounding of the floating-point bounds. Interval arithmetic (shortened hereafter to IA) enjoys some very important properties:

- IA ensures completeness, such that no solution is ever lost. Consequently, any computation that leads to empty intervals for the domain of the variables involved is a *proof* for the non-existence of solutions;
- many classical numerical algorithms, such as Gauss-Seidel's or Newton's, have been adapted to intervals, thereby benefiting from new properties. For example, the interval Newton algorithm has been proved to be convergent whenever the starting domains contain a solution. What is more, condition criteria—based on Miranda's and Brouwer's theorems—may be

used to prove the existence and uniqueness of a solution in a computed interval;
- IA permits reasoning on sets of values, thus allowing the devising of efficient procedures for nonlinear global optimization.

Proper implementation of interval arithmetic demands floating-point numbers properties and rounding facilities that are available from all IEEE 754 [2] compliant computers—the great majority of them—since 1984. However, despite the above-mentioned qualities of interval arithmetic, there still does not exist any language providing intrinsic support for it—a mandatory condition for performance and correctness issues. Are the proposals for integrating interval support into Fortran 2000[a] and Java[3] a harbinger of change? At least not for a widely used language such as C++, whose adopted standard does not state anything about IA, while defining a generic class for complex arithmetic (*i.e.* a template of class that can be instantiated with different float formats, depending on the required precision). Some C++ interval libraries already exist [4,5] though they do not respect the spirit of standard C++ libraries since they do not provide generic classes but a "hard-wired" class for intervals whose bounds are usually `doubles`.

In this paper, we present the design and implementation of JAIL[b], a templated C++ interval library, and we exhibit the benefits obtained from parameterization, namely: a runtime speed-up over non-templated libraries; a seamless integration into the C++ *Standard Template Library*; an enhanced reliability and portability over different platforms; a better versatility, by allowing the use of different floating-point formats for the interval bounds, depending on the required precision. The impact of some implementation tricks is also investigated.

The rest of the paper is organized as follows: basics of interval arithmetic are briefly presented in Section 2; C++ techniques to be used in designing JAIL are described in Section 3; Section 4 then introduces JAIL and presents some of the techniques used to improve both its speed and correctness; some experimental results are given and analysed in Section 5; last, directions for future researches are discussed in Section 6.

2 Interval arithmetic

An interval is a connected set of reals. While it may be closed or open, we will only consider closed intervals in the following, that is sets of the form:

[a] See `http://interval.usl.edu/F90/f96-pro.asc`
[b] JAIL stands for "*Just Another Interval Library*".

$I = [a \,..\, b] = \{c \in \mathbb{R} \mid a \leqslant c \leqslant b\}$, where a and b must be floating-point numbers for intervals to be stored and manipulated on computers. Floating-point arithmetic features that are relevant to interval arithmetic are presented in Section 2.1 while interval arithmetic is surveyed in Section 2.2. Due to lack of space, both topics are but sketched in the following sections. The reader is referred to Goldberg's survey [6] and Moore's book [1] for thorough presentations of floating-point arithmetic and interval arithmetic.

2.1 Floating-point arithmetic

The finite nature of computers precludes them from representing the whole set $\mathbb{R}$ of real numbers. Modern computers handle floating-point numbers (or simply, *floats*) of various precisions, whose formats are precisely defined by the IEEE 754 standard [2]. A set $\mathbb{F} \subset \mathbb{R}$ of floats in one of these given precisions must be closed under all supported operations in order not to break the computation stream. Practical consequences are:

- the compactification of $\mathbb{F}$ with *two infinities* ($-\infty$ and $+\infty$) with the rules: $\forall a \in \mathbb{F}, a > 0\colon \quad a/0 = +\infty$ and $-a/0 = -\infty$. By symmetry, the IEEE 754 standard defines two zeroes (-0 and $+0$, with $-0 = +0$), such that: $\forall a \in \mathbb{F}, a > 0\colon \quad a/+\infty = +0$ and $a/-\infty = -0$. In the sequel, let $\mathbb{F}^\infty$ be the set $\mathbb{F} \cup \{-\infty, +\infty\}$;
- the introduction of special "numbers", *NaNs*[c], that are the result of any otherwise undefined operation, such as extracting the square root of a negative number, or divisions like $0/0$ and $+\infty/+\infty$;
- the specification of *rounding operations*: given $x \in \mathbb{R} \setminus \mathbb{F}$ a real number, the IEEE 754 defines four policies for rounding x to a float: rounding towards $+\infty$, $-\infty$, 0, and the nearest-even. The only ones of interest to us are rounding towards $+\infty$ and towards $-\infty$ that can be formally defined by:

$$\text{Rounding towards } -\infty\colon \quad \lfloor\ \rfloor : \mathbb{R} \longrightarrow \mathbb{F}^\infty, \quad x \longmapsto \max\{y \in \mathbb{F}^\infty \mid y \leqslant x\}$$
$$\text{Rounding towards } +\infty\colon \quad \lceil\ \rceil : \mathbb{R} \longrightarrow \mathbb{F}^\infty, \quad x \longmapsto \min\{y \in \mathbb{F}^\infty \mid x \leqslant y\}$$

  The IEEE 754 standard also stipulates that the real result of any of the five operations $\{+, -, \times, \div, \sqrt{}\}$ must be *correctly rounded*, that is rounded by excess or by default to the nearest floating-point number. Due to the *Table maker's dilemma* [6], no requirement of the sort is imposed on transcendental functions, whose precision is thus implementation-dependent.

[c] NaN stands for "*Not A Number*".

2.2 Real and floating-point intervals

In this section, we will consider both the *real interval set* $\mathbb{I}_{\mathbb{R}}$ (*i.e.* intervals whose bounds are real numbers) and the *floating-point interval set* $\mathbb{I}_{\mathbb{F}}$ (*i.e.* intervals whose bounds are floats).

Real intervals. Real interval arithmetic is defined as follows: given $I_1 = [a \,..\, b]$ and $I_2 = [c \,..\, d]$ two intervals in $\mathbb{I}_{\mathbb{R}}$, $\circ \in \{+, -, \times, \div\}$, and $\varphi \in \{\sqrt{}, \exp, \ln, \cos, \dots\}$, real intervals $I_3 = I_1 \circ I_2$ and $I_4 = \varphi(I_1)$ are defined by $I_3 = \{x \circ y \mid x \in I_1,\, y \in I_2\}$ and $I_4 = \{\varphi(x) \mid x \in I_1\}$. From monotony considerations, one can easily deduce the following formulae from the above definitions:

$$
\begin{aligned}
[a \,..\, b] + [c \,..\, d] &= [a + c \,..\, b + d], \qquad [a \,..\, b] - [c \,..\, d] = [a - d \,..\, b - c] \\
[a \,..\, b] \times [c \,..\, d] &= [\min(ac, ad, bc, bd) \,..\, \max(ac, ad, bc, bd)] \\
[a \,..\, b] \div [c \,..\, d] &= [\min(a/c, a/d, b/c, b/d) \,..\, \max(a/c, a/d, b/c, b/d)],\ 0 \notin [c \,..\, d] \\
\exp([a \,..\, b]) &= [\exp(a) \,..\, \exp(b)]
\end{aligned}
$$

Note that division may also be defined such that dividing by an interval spanning 0 is permitted. See Kahan's *extended interval arithmetic* [7].

A weakness of interval arithmetic is that it is only *sub-distributive*: $\forall I_1, I_2, I_3 \in \mathbb{I}_{\mathbb{R}} \colon I_1 \times (I_2 \pm I_3) \subseteq I_1 \times I_2 \pm I_1 \times I_3$. On the other hand, its strength comes from the following theorem.

Theorem 1 (IA fundamental theorem [1]) *Given $F(X_1, \dots, X_n)$ a rational expression on intervals $X_1, \dots, X_n$, the following does hold for all $X'_1, \dots, X'_n \in \mathbb{I}_{\mathbb{R}}$: $X'_1 \subseteq X_1, \dots, X'_n \subseteq X_n \Rightarrow F(X'_1, \dots, X'_n) \subseteq F(X_1, \dots, X_n)$*

Consequently, it is possible to compute the range of a rational expression over a given domain by merely evaluating its extension over intervals.

Example 1 (Computing the range of a function) *Given the function $f(x) = (x + 1/2)^2 - 1/4$ and the interval $I = [-1 \,..\, 1]$, computing the range of f over I is as easy as evaluating the interval expression $I + [1/2 \,..\, 1/2])^2 - [1/4 \,..\, 1/4]$, which leads to $f(I) = [-1/4 \,..\, 2]$. Due to sub-distributivity, the form considered for evaluating f over intervals is important for tightness of the result. For example, evaluating f as $x(x+1)$ would lead to $f(I) = [-2 \,..\, 2]$.*

Floating-point intervals. In order to effectively compute with intervals, one has to make the shift from real intervals to floating-point intervals. This is achieved by *outward rounding* of real bounds to float bounds in order to ensure completeness. For example, the preceding definitions become:

$$
\begin{aligned}
[a \,..\, b] + [c \,..\, d] &= [\lfloor a + c \rfloor \,..\, \lceil b + d \rceil] \qquad [a \,..\, b] - [c \,..\, d] = [\lfloor a - d \rfloor \,..\, \lceil b - c \rceil] \\
&\dots \qquad\qquad \exp([a \,..\, b]) = [\lfloor \exp(a) \rfloor \,..\, \lceil \exp(b) \rceil]
\end{aligned}
$$

Due to IEEE 754 correct rounding requirement for $+, -, \times, \div, \sqrt{}$, the computed intervals for these operations are the smallest representable. This is not so for the other operators such as exp, whose precision is not specified. As a consequence, $\lfloor \exp(a) \rfloor$ may be a value greater than the true result, depending on the accuracy of the exp function on a given platform. In order to overcome this problem, implementors of interval libraries usually rely on *epsilon-inflation*: a small quantity is added or subtracted from the value of $\exp(a)$, depending on the intended rounding direction. However, the preceding formulae cannot be directly used for implementing floating-point interval arithmetic. For example, consider intervals $I_1 = [-\infty \,..\, 3]$ and $I_2 = [-1 \,..\, 0]$. Applying blindly the formula for the product of I_1 and I_2 leads to:

$$\begin{aligned} I_1 \times I_2 = [\ &\min(\lfloor -\infty \times -1 \rfloor, \lfloor -\infty \times 0 \rfloor, \lfloor 3 \times -1 \rfloor, \lfloor 3 \times 0) \rfloor \,.. \\ &\max(\lceil -\infty \times -1 \rceil, \lceil -\infty \times 0 \rceil, \lceil 3 \times -1 \rceil, \lceil 3 \times 0) \rceil)] \end{aligned}$$

IEEE 754 specifies that $-\infty \times 0$ must be a NaN. Since NaNs are unordered, the result of $I_1 \times I_2$ depends on the implementation of min and max functions. It is then crucial to avoid the arising of NaNs by testing bound values prior to float multiplication.

3 C++ techniques

The C++ techniques that are used in the design of JAIL are now presented in this section. Generic programming through *templated classes* is first introduced; the "traits [8]" method for defining interfaces much like *signatures* is then described.

3.1 Templated classes

A C++ templated class is a pattern for an actual class that is parameterized by a type. It is not a valid class until instantiated. For example, one can instantiate a templated class `list` to define lists of integers `list<int>` or lists of characters `list<char>`. The whole C++ standard library is composed of templated classes (containers such as lists, hash tables, ...) and templated functions (sorting procedures,...). Parameterization through templated class allows factorizing code for many types. It may however lead to weird situations. For example, consider Prog. 1 for a C++ class `MyNumber` whose intended use is to provide a means to count the number of calls to arithmetic operators. The class is parameterized by the type `T` of the actual values to be manipulated.

Execution of Prog. 1 will print values 0.70867 (*i.e.* cos(5.5)) and 0 on the screen, where the 0 comes from the fact that the call to `cos` for `b` induces

the promotion of the integer value 6 to a `double` for calling the "`double cos(double)`" function of the standard library at line 18, and then a cast of the result back to an `int` for returning a `MyNumber<int>` object.

PROG. 1. The `MyNumber` class without "traits"

```
1 template<class T>
2 class MyNumber {
3 public:
4   MyNumber(const T& x) :  val(x) { }
5   friend MyNumber
6     cos<>(const MyNumber& x);
7 private:
8   T val;
9   static unsigned int nbCallsCos;
10 };
11
12 template<class T> unsigned int
13         MyNumber<T>::nbCallsCos=0;
14
15 template<class T> MyNumber<T>
16 cos(const MyNumber<T>& x) {
17   ++MyNumber<T>::nbCallsCos;
18   return MyNumber<T>(::cos(x.val));
19 }
20
21 int main() {
22   MyNumber<double> a(5.5);
23   MyNumber<int> b(6);
24
25   cout << cos(a) << cos(b) << endl;
26 }
27
```

One would indeed like to have the ability to return a `MyNumber<double>` object from a `MyNumber<int>` for some operators, and specify which function `cos()` to use depending on type `T`. The "traits" technique devised by Myers [8] is an elegant solution to these requests.

3.2 The "traits" technique

A *trait* is a templated class collecting types and method names acting as an interface. Prog. 2 presents the `MyNumber` class implemented with traits. The `traits` class defines a type `Real` and a `cos()` function whose input and output have the same type by default. Lines 6 through 11 present a specialization of the trait for `short int`s. The `MyNumber` class then uses the `traits` class to know what should be the `cos()` function to use and what is the type of its result, depending on its parameter `T`. As a consequence, Prog. 2 prints the value 0.96017 (*i.e.* cos(6)), since the `cos()` function for `MyNumber<short>` knows that it must invoke function "`float cos(short)`" and return a `MyNumber<float>` as a result.

The traits technique then permits optimizing the code by specializing the class for some types (for example, we use here a "`float cos(short)`" that should hopefully be faster than the standard "`double cos(double)`" function. Reliability is also enhanced since the user must explicitly specify which functions to call in the `MyNumber` methods, thereby avoiding ambiguities.

4 Design and implementation of JAIL

The heart of the JAIL library is the templated class `Interval`, that contains most of the interval operators and relations defined in the *Basic Interval*

Arithmetic Specification [9].

PROG. 2. The `MyNumber` class with "traits"

```
template<class T> struct traits {
  typedef T Real;
  static T cos(T x);
};

template<> struct traits<short> {
  typedef float Real;
  static float cos(short x) {
    return ::cos(float(x));
  }
};

template<class T> class MyNumber {
public:
  typedef traits<T>::Real
     Real;
  MyNumber(const T& x) :  val(x) { }
  friend MyNumber<Real>
       cos<>(const MyNumber<T>& x);
private:
  T val;
  static unsigned int nbCallsCos;
};

template<class T>
MyNumber<typename traits<T>::Real>
cos(const MyNumber<T>& x) {
  ++MyNumber<T>::nbCallsCos;
  return MyNumber<typename
    traits<T>::Real>(
            traits<T>::cos(x.val));
}

int main() {
  MyNumber<short> b(6);
  cout << cos(b) << endl;
}
```

The `Interval` class may be instantiated with any floating-point format by specializing the `JailRealTraits` trait (see Prog. 3) that specifies all constants and functions used in implementing interval functions. In particular, the trait requires specifying downward and upward rounded versions for each function whose accuracy is not specified by the IEEE 754 standard (*e.g.* `cosDn()` and `cosUp()`). Besides enhancing the versatility of the library by permitting generic programming with intervals, the use of traits also increases its reliability when porting it on different platforms by making explicit the need for correctly rounded functions. Depending on the possibilities at hand on a given platform, one can define a function such as `cosDn()` by using epsilon-inflation or by directly calling a `cos()` function in a mathematical library in which it is known to be correctly rounded. What is more, the amount of epsilon-inflation may thus be easily parameterized according to the precision of the available functions without modifying the `Interval` class itself.

PROG. 3. JAIL's trait specialized for the `double` type

```
struct JailRealTraits<double> {
 typedef double JailReal;
 static inline double NextFloat(double x);
 static const double MAXREAL;
 static inline double cosDn(double x);
 static inline double cosUp(double x);
 ...
};
```

Modern FPUs are usually pipe-lined: at one time, the FPU contains several instructions at different decoding stages. Modifying the rounding direction for computing interval bounds requires flushing all these instructions, then modifying a flag before admitting new instructions in the pipe-line. Such an interruption of the stream greatly lessens performances. In order to over-

come this problem, JAIL uses only rounding towards $+\infty$ by relying on the following IEEE 754 guaranteed property:

$$\lfloor x \circ y \rfloor = -\lceil (-x) \circ y \rceil, \quad \text{for } \circ \in \{+, -, \times, \div\}$$

For the square root, we use the following property (the proof is easily done by using the definition of $\lceil\ \rceil$ twice):

$$\forall x \in \mathbb{F}, x < 0\colon\ -\left\lceil \frac{x}{\lceil \sqrt{-x} \rceil} \right\rceil \leqslant \lfloor \sqrt{-x} \rfloor$$

In order to avoid using the negation twice, we have chosen to directly represent the left bound of an interval by its opposite (*e.g.* the interval $[-6..4]$ is internally represented by the pair $\langle 6, 4 \rangle$. The code for computing the sum of two intervals is then as follows:

```
Addition(in I1, I2; out I3 = I1 + I2)
begin
  RoundUp()
  I3 ← −(I1 + I2)
  I3 ← I1 + I2
end
```

One thus halves the number of rounding direction switching. But one can do better still, provided that each function that modifies the rounding direction (basically, output routines) reset it to "towards $+\infty$". It is then possible to switch the rounding towards $+\infty$ only once at the beginning of the computation. This is what we call the *trust on* mode since the library trusts functions for preserving the rounding direction. By contrast, the *trust off* mode requires switching the rounding towards $+\infty$ for each interval operation.

NaNs propagate along all the computation, as the empty interval do. We have then decided to use $[NaN\ ..\ NaN]$ for representing the empty interval (actually, suffices for one of the bounds to be a NaN). Since NaNs are not comparable, it is then necessary to negate tests such as test for emptiness:

```
isEmpty(in: I1; out: I1 == ∅)
begin
  return (¬(−I1 ⩽ I1))
end
```

The test $(-\underline{I_1} \leqslant \overline{I_1})$ is false whenever one of the bounds is a NaN or the right bound is smaller than the left bound.

Since NaNs are only characterized by the the value of the exponent (suffices for the mantissa to be different from 0), one can use the bits of their mantissa for coding information recording when and why an empty interval appeared.

5 Experimental results

In order to validate our implementation choices, we have developed five versions of our library: a templated version using the *trust on* mode and another using the *trust off* mode, and three non-templated versions (one in *trust on* mode, the second in *trust off* mode, and the last using downward rounding for computing left bounds and upward rounding for right bounds). We have then used a set of intervals exercising the whole code of the operators tested. Table 1 presents the computation times for this benchmark. One may easily see that the templated version using the *trust on* mode is the fastest of all. However, the compiling time suffers greatly from the use of templates (this is a well known drawback that should be made less accurate in the future with new generations of C++ compilers).

Table 1. Impact of the implementation choices for JAIL

	Parameterized version		Non-parameterized version		
	`trust on`	`trust off`	`t. on`	`t. off`	`Dn/Up`
$x + y$	19204	24175	34445	41281	49280
$x - y$	17526	24812	34543	41311	42840
$x \times y$	17220	22559	29709	35383	38121
$x \div y$	33105	33563	39561	40634	41792
$\sqrt{x}$	12829	14451	21412	22942	26693
x^2	7309	9378	13827	15778	18071
$x^{-10} \dots x^{10}$	13141	13913	15997	16572	11632
e^x	23915	23182	26262	24810	24545
$\log x$	18404	17934	21069	20769	20860
$\cos x$	19218	19577	19951	20421	20766
$\sin x$	21289	21628	22220	22581	32663
$\tan x$	14807	14970	15660	15608	16013
compilation	11200	13300	4200	4100	4000

Times in milliseconds on a SUN UltraSparc 1/167 MHz with egcs 2.91.66

We have also compared JAIL with other available (non-templated) interval libraries, using the same benchmark as above. The results are summarized in Table 2. They show that the higher versatility of JAIL (genericity of the `Interval` class) does not prevent it from achieving good performances since it is the fastest of all the libraries tested on the elementary operations. There is still room for improvement as for the trigonometric functions, though. Note also that Profil is the fastest for computing powers, but it returns wider intervals (use of an adaptation of the *Russian peasant method* to exponentiation).

6 Conclusion

We have shown that a generic interval library for C++ can be obtained at no extra cost as for the computation time, thereby permitting to seamlessly

integrate it into the standard C++ library. What is more, the use of templates and traits enhances the reliability of the resulting library.

Future works on JAIL include implementing the whole *Basic Interval Arithmetic Specification* [9], and integrating *directed interval arithmetic* [10], a variant of interval arithmetic that permits computing inner-approximations of the range of functions.

Table 2. Comparing interval arithmetic libraries on an UltraSparc 1/167 MHz

	Profil	RVInterval	fi_lib	JAIL†
$x + y$	26362	53400	48479	19204
$x - y$	26317	53102	47637	17526
$x \times y$	25951	40274	35758	17220
$x \div y$	35521	38245	39538	33105
$\sqrt{x}$	321568	12382	23853	12829
x^2	16426	26819	14133	7309
$x^{-10} \dots x^{10}$	6698	25747	—	13141
e^x	201797	23599	11605	23915
$\log x$	119980	20195	10940	18404
$\cos x$	109844	7963	6295	19218
$\sin x$	91883	16982	5953	21289
$\tan x$	41657	—	4117	14807

Results in milliseconds with egcs 2.91.66

† *Parameterized version with* `trust on`

References

1. R. E. Moore. *Interval Analysis*. Prentice-Hall, 1966.
2. IEEE. IEEE standard for binary floating-point arithmetic. Ieee std 754-1985, Instit. Electrical and Electronics Engineers, 1985.
3. J. D. Darcy. *Borneo 1.0.2: Adding IEEE 754 floating point support to Java*. Univ. California, Berkeley, May 1998.
4. O. Knüppel. PROFIL/BIAS—a fast interval library. *Computing*, 53:277–287, 1996.
5. W. Hofschuster and W. Kraemer. A fast public domain interval library in ANSI C. In *Procs. 15th IMACS World Congress on Scientific Computation, Modelling and Applied Maths.*, volume 2 of *Num. Maths.*, pages 395–400, 1997.
6. D. M. Goldberg. What every computer scientist should know about floating-point arithmetic. *ACM Comp. Surveys*, 23(1):5–48, March 1991.
7. W. M. Kahan. A more complete interval arithmetic. Tech. report, Univ. toronto, 1968.
8. N. C. Myers. A new and useful technique: "traits". *C++ Report*, 7(5):32–35, June 1995.
9. D. Chiriaev and G. W. Walster. Interval arithmetic specifications. Manuscript J3/97-199 for ANSI X3J3, July 1997.

10. S. M. Markov. On directed interval arithmetic and its applications. *J. Univ. Comp. Sci.*, 1(7):514–526, 1995.